Controlled Physical Unknown Functions: Applications to Secure Smartcards and Certified Execution

Blaise Gassend, Dwaine Clarke,
Marten van Dijk, Srinivas Devadas

June 10, 2002

Abstract

A Physical Unknown Function (PUF) is a function that is easy to evaluate but hard to characterize. We introduce controlled physical unknown functions (CPUFs) which are PUFs that can only be accessed via an algorithm that is physically bound to the PUF in an inseparable way.

Controlled PUFs enable several applications including certified execution, where a certificate is produced that proves that a specific computation was carried out on a specific processor. Certified execution has many benefits, including protection against malicious volunteers/entities in grid computing, anonymous computing and other forms of distributed computation.

An integrated circuit (IC) containing a controlled PUF can be authenticated using challenge-response pairs (CRP's). We describe protocols for CRP management that protect against a man-in-the-middle attack. We also describe protocols through which controlled PUF's can be used in the applications of smartcard identification and certified execution, and briefly discuss a software licensing application.

1 Introduction

A Physical Unknown Function (PUF) is a function that is easy to evaluate but hard to characterize. PUFs can be implemented in different ways and can be used in identification and authentication applications [GCvDD02, Rav01]. In this paper, we introduce controlled physical unknown functions (CPUFs) which are PUFs that can only be accessed via an algorithm that is physically bound to the PUF in an inseparable way.

PUFs and controlled PUFs enable a host of applications, including smartcard identification, certified execution and software licensing. In current smartcards, it is possible for someone who is in possession of a smartcard to produce a clone of it, by extracting its digital key information through one of many well documented attacks [And01]. With a unique PUF on the smartcard that can be used to authenticate the chip, a digital key is not required: the smartcard *hardware* is itself the secret key. This key cannot be duplicated, so a person can lose control of it, retrieve it, and continue using it.

Certified execution produces a certificate which proves that a specific computation was carried out on a specific processor chip, and that the computation produced a given result. The person requesting the computation can then rely on the trustworthiness of the chip manufacturer who can vouch that he produced the chip, instead of relying on the owner of the chip, who could make up the result without actually executing the computation.[1] Certified execution is very useful in grid computing (e.g., SETI@home) and other forms of distributed computation to protect against malicious volunteers. In fact, certified execution can enable a business model for anonymous computing, wherein computation can be sold by individuals and the customer can be ensured reliability of service, via the generation of certificates.

Controlled PUFs can also be used to ensure that a piece of code only runs on a processor chip that has a specific identity defined by a PUF. In this way, pirated code would fail to run.

In this paper, we will describe the implementation of controlled PUFs and key management for controlled PUFs. We will focus on the applications of smartcard identification and certified execution and describe the protocols for these applications. We will only briefly touch upon the software licensing application.

We define physical unknown functions (PUFs) and controlled PUFs in Section 2. The reader who is not interested in PUF or CPUF implementations can then skip to Section 5. Implementation of PUFs and controlled PUFs on silicon integrated circuits is the subject of Section 3. We also describe how PUFs can

[1] Of course, the person requesting the computation could attempt to verify the result produced by either running the computation on a trusted computer or requesting another chip owner to run the computation. Both alternatives are not efficient.

be strengthened using associated control in Section 4.

In Section 5, we describe our model for using controlled PUFs. In Section 6, we describe a man-in-the-middle attack, and the protocols that protect a PUF from it.

We describe how controlled PUFs can be applied to authentication and certified execution problems in Section 7, and briefly describe a software licensing application. We conclude the paper in Section 8.

2 Definitions

Definition 1 *A Physical Unknown Function (PUF) is a function that maps challenges to responses, that is embodied by a physical device, and that verifies the following properties:*

1. *Easy to evaluate: The physical device is easily capable of evaluating the function in a short amount of time.*
2. *Hard to characterize: From a polynomial number of plausible physical measurements (in particular, determination of chosen challenge-response pairs), an attacker who no longer has the device, and who can only use a polynomial amount of resources (time, matter, etc...) can only extract a negligible amount of information about the response to a randomly chosen challenge.*

In the above definition, the terms short and polynomial are relative the size of the device, which is the security parameter. In particular, short means linear or low degree polynomial. The term plausible is relative to the current state of the art in measurement techniques and is likely to change as improved methods are devised.

In previous literature [Rav01] PUFs were referred to as Physical One Way Functions, and realized using 3-dimensional micro-structures and coherent radiation. We believe this terminology to be confusing because PUFs do not match the standard meaning of one way functions [MvOV96].

Definition 2 *A PUF is said to be Controlled if it can only be accessed via an algorithm that is physically linked to the PUF in an inseparable way. In particular this algorithm can restrict the challenges that are presented to the PUF and can limit the information about responses that is given to the outside world.*

Control turns out to be the fundamental idea that allows PUFs to go beyond simple authenticated identification applications. How this is done is the main focus of this paper.

Definition 3 *A type of PUF is said to be Manufacturer Resistant if it is technically impossible to produce two identical PUFs of this type given only a polynomial amount of resources.*

Manufacturer resistant PUFs are the most interesting form of PUF as they can be used to make unclonable systems.

Definition 4 *A PUF is said to be One Way if it is a collision resistant one way function.*

In practice, the types of functions that are good PUF candidates are often also one way. Some algorithms can be simplified when this is the case.

3 Implementing a Controlled Physical Unknown Function

In this section, we describe ways in which PUFs and CPUFs could be implemented. In each case, a silicon IC enforces the control on the PUF.

3.1 Digital PUF

It is possible to produce a PUF with classical cryptographic primitives. If an IC is equipped with a secret key k, and a one-way hash function h, and tamper resistant technology is used to make k impossible to extract from the IC, then the function

$$x \rightarrow h(k, x)$$

is a PUF. If control logic is embedded on the tamper resistant IC along with the PUF, then we have effectively created a CPUF.

However, this kind of CPUF is not very satisfactory. First, it requires high quality tamper-proofing. There are systems available to provide such tamper-resistance. For example, IBM's PCI Cryptographic Coprocessor, encapsulates a 486-class processing subsystem within a tamper-sensing and tamper-responding environment where one can run security-sensitive processes [SW99]. Smart cards also incorporate barriers to protect the hidden key(s), many of which have been broken [And01]. In general, however, effective tamper resistant packages are expensive and bulky.

Secondly, the digital PUF is not manufacturer resistant. The PUF manufacturer is free to produce more than one IC with the same secret key, or someone who manages to violate the IC's tamper-resistant packaging and extract the secret key can easily produce a clone of the PUF.

Because of these two weaknesses, a digital PUF does not offer any security advantage over conventional cryptographic primitives, and it is therefore better to use a conventional crypto-system.

3.2 Silicon PUF

3.2.1 Statistical Variation of Delay

By exploiting statistical variations in the delays of gates and wires within the IC, we can create a manufacturer resistant PUF [GCvDD02]. Manufactured IC's, from either the same lot or wafer have inherent delay variations. There are random variations in dies across a wafer, and from wafer to wafer due to, for instance, process temperature and pressure variations, during the various manufacturing steps. The magnitude of delay variation due to this random component can be 5% or more for metal wires, and is higher for devices.

On-chip measurement of delays can be carried out with very high accuracy, and therefore the signal-to-noise ratio when delays of corresponding wires across two or more IC's are compared is quite high. The delays of the set of devices in a circuit is unique across multiple IC's implementing the same circuit with very high probability, if the set of devices is large [GCvDD02]. These delays correspond to an implicit hidden key, as opposed to the explicitly hidden key in a digital PUF. While environmental variations can cause changes in the delays of devices, relative measurement of delays, essentially using delay ratios, provides robustness against environmental variations, such as varying ambient temperature, on-chip junction temperature, and power supply variations.

3.2.2 Challenge-Response Pairs

Given a PUF, challenge-response pairs can be generated, where the challenge can be a digital input stimulus, and the response depends on the transient behavior of the PUF, and can be a precise delay measure, or a digital response based on measured delay. The number of potential challenges grows exponentially with the number of inputs to the IC. Therefore, while two IC's may have a high probability of having the same response to a particular challenge, if we apply enough challenges, we can distinguish between the two IC's.

Upon every successful authentication of a given IC, a set of challenge-response pairs is potentially revealed to an adversary. This means that the same challenge-response pair cannot be used again. If the adversary can learn the entire set of challenge-response pairs, he can create a model of a counterfeit IC. However, the number of possible challenge-response pairs is exponentially large. Since an exponentially large set cannot be stored, one plausible approach is to "recharge" the set of stored challenge-response pairs periodically, by turning in the IC to the authority that performs the authentication.

As before, if control logic is embedded on the IC along with the PUF, then we have effectively created a CPUF. In our protocols described in Section 6, challenge-response pairs can be reused for a CPUF because the response is never sent in the clear.

3.2.3 Attacks on Silicon PUFs

There are many possible attacks on manufacturer resistant PUF's – duplication, model building using direct measurement, and model building using adaptively-chosen challenge generation. We briefly discuss these and show that significant barriers exist for each of these attacks. A more detailed description can be found in [GCvDD02].

The adversary can attempt to duplicate a PUF by fabricating a counterfeit IC containing the PUF. However, due to statistical variation, unless the PUF is very simple, the adversary will have to fabricate a huge number of IC's and precisely characterize each one, in order to create and discover a counterfeit.

Assume that the adversary has unrestricted access to the IC containing the PUF. The adversary can attempt to create a model of the IC by measuring or otherwise determining very precisely the delays of each device and wire within the IC. Direct measurement of device delays requires the adversary to open the package of the IC, and remove several layers, such as field oxide and metal. One can also create a package which has a significant effect on the delays of each device within the IC, and the removal of the package will immediately destroy the PUF, since the delays will change appreciably.

The adversary could try to build a model of the PUF by measuring the response of the PUF to a polynomial number of adaptively-chosen challenges.[2] We believe this to be the most plausible form of attack. However, there is a significant barrier to this form of attack as well because creating timing models of a circuit accurate to within measurement error is a very difficult problem that has received a lot of attention from the simulation community. Manageable-sized timing models can be produced which are within 10% of the real delays, but not within the measurement accuracy of $\approx 0.1\%$.

4 Improving a PUF Using Control

Now that we have seen what a CPUF is, and how it can be implemented, we shall start to look at some simple advantages of CPUFs over non-controlled PUFs. This section shows how the control that is

[2] Clearly, a model can be built by exhaustively enumerating all possible challenges, but this is intractable.

placed around a PUF can be used to overcome a number of imperfections in the PUF.

In each case, we have a PUF f that we are trying to improve in some way. Control allows us to improve f by constructing a new PUF g, that is based on f. The control only allows f to be evaluated as part of an evaluation of g, and only uses the result of the evaluation of f to help evaluate g.

The block diagram in figure 1 shows most of the improvements that are discussed in this section. The reader can refer to them to get a better understanding of what is being explained.

4.1 Preventing Chosen Challenge Attacks

Unless one ventures into quantum effects (which would make a PUF highly unreliable), the number of physical parameters that define a PUF is proportional to the size of the system that defines it. Therefore, in principle, if an attacker is able to determine a number of primitive parameters that is proportional to the size of the physical system, he can use them to simulate the system and thus clone the PUF.

To try to determine primitive parameters, the attacker gets a number of challenge-response pairs (CRPs), and uses them to build a system of equations that he can try to solve. By definition, for a PUF, these equations are impossible to solve in reasonable time. However, there can be physical systems for which most CRPs lead to unsolvable equations, while a small subset of CRPs give equations that are able to break the PUF (which consequently is not really a PUF). Such a system is not secure because an adversary can use the CRPs that lead to simple equations to get a solvable system of equations, calculate the primitive parameters, and clone the PUF by building a simulator.

With control, it is nevertheless possible to build a secure system out of one of these broken PUFs. One way of doing this is for the control layer to simply refuse to give responses to challenges that lead to simple equations. Unfortunately, this method assumes that we know all the strategies that the attacker might use to get a simple set of equations from a chosen set of CRPs.

We can do even better if we pre-compose the broken PUF with a one way function. Instead of using f directly, we use

$$g(x) = f(h(x)),$$

where h is a one-way function. With this method, it is impossible for the adversary to choose the challenge $h(x)$ that is being presented to the underlying PUF, so even if he finds a challenge that would break it, he is unable to present that challenge. Now, there is no need for the designer of the PUF to know what challenges the adversary might try to exploit.

4.2 Post-Composition with a One-Way Function

It is desirable for the output of a PUF to exhibit as much randomness as possible to prevent an adversary from guessing the response to one challenge by using the response to another challenge. However, the output of a physical system is likely to produce similar responses when faced with similar stimuli. Moreover, as we discussed in section 4.1, CRPs can be used to get systems of equations that relate the PUF's underlying physical parameters.

Both of these risks can be eliminated by doing a simple transformation on the PUF. If f is the PUF that we are trying to improve, and h is a one-way hash function, then

$$g(x) = h(x, f(x))$$

is a stronger PUF. With this method, we can take a PUF that has good properties such as manufacturer resistance, and make it into a PUF that has the advantages of a digital PUF. The one-way hash function's avalanche-effect ensures that nearby outputs of f will lead to completely different outputs of the composite function, and the one-way nature of h means that to set up a system of equations, the adversary has to invert h (or include the definition of h in the system of equations, which is just as bad).

4.3 Unique Identifier

With manufacturer resistant PUFs, the manufacturer resistance is typically a result of the manufacturer's limited control over process variations. Each PUF is different because of these variations. However, it is possible that there will be identical PUFs. This isn't much of a problem, because in general finding a pair of PUFs that is identical requires producing, and comparing an unreasonable number of PUFs.

Nevertheless, it is possible to guarantee that any two PUFs are different. To do so, we combine the actual challenge and a unique identifier that is unique to the chip with a hash before running them through the rest of the PUF. The unique identifier that is used here need not be secret, and can be the IC's serial number, for example.

In this way, no two PUFs are identical, and even if two CPUFs share the same underlying PUF f, there is no way for an adversary to find this out (the manufacturer might be able to discover it before setting the PUF's unique identifier, but the cost of testing is prohibitive in any case).

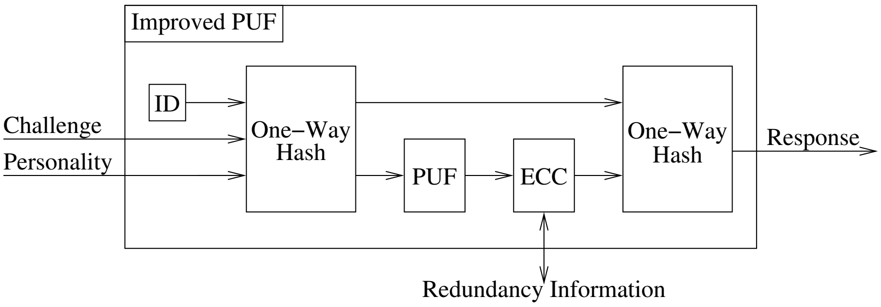

Figure 1: This diagram shows how control can be used to improve a PUF. One-way hash functions are used at the input and output of the PUF, an Error Correcting Code is used to make the PUF reliable, a unique identifier guarantees that no two PUFs will be identical, and a personality selector allows the owner of the PUF to maintain his privacy.

4.4 Giving a PUF Multiple Personalities

A possible concern with the use of PUFs is in the area of privacy. Indeed, past experience shows that users feel uncomfortable with processors that have unique identifiers, because they feel that they can be tracked. PUFs being a form of unique identifier, users could have the same type of concern with their use.

This problem can be solved by providing a PUF with multiple personalities. The owner of the PUF has a parameter that she can control that allows her to show different facets of her PUF to different applications. To do this, we hash the challenge with a user-selected personality number, and use that hash as the input to the rest of the PUF.

In this way, the owner effectively has many different PUFs at her disposal, so third parties to which she has shown different personalities cannot determine if they interacted with the same PUF.

Section 6.4 goes into the details of the protocols that use multiple personalities.

4.5 Error Correction

In many cases, the PUF is being calculated using an analog physical system. It is to be expected that slight variations from one run to the next will cause slight changes in the digitized output of the PUF. This means that the chip only produces an approximation of the response that is expected of it. The chip and the challenger cannot directly compare the real response with the desired response as this would require sending one of the responses in the clear, thus compromising the shared secret. Therefore, something must be done to make the PUF's output consistent.

A suitably selected error correcting code is one possibility. When a challenge-response pair is created, some redundant information is also produced that should allow slight variations in the measured parameters to be corrected for. On subsequent uses of the challenge-response pair, the redundant information is provided to the PUF along with the challenge. It is used to correct the response from the physical system.

Naturally, the error correction must take place directly on the measured physical parameters. In particular, if any one-way functions are added to improve the PUF, they should not be added between the physical measurements and the error correction.

4.6 Multiple Rounds

To add even more complexity to the attacker's problem, it would be possible to use the PUF circuit multiple times to produce one response. The corrected response from one round can be fed back into the PUF circuit. After a few rounds have been done, all their outputs could get merged together along with the challenge, the personality and the chip's identifier and passed through a one-way hash function to produce the response.

5 Models

5.1 Application Model

Figure 2 illustrates the basic model for applications using the PUF.

- The user is the principal that wants to make use of the computing capabilities of a chip.
- The user and the chip are connected to one another by an untrusted public communication channel.
- The interface between the chip and the untrusted communication channel is a PUF.

- Given a challenge a PUF can compute a corresponding response.

- The user is in the possession of her own private list of CRPs originally generated by the PUF. The list is private because only the user and the PUF know the responses to each of the challenges in the list. We assume that the user's challenges can be public, and that the user has established several CRPs with the PUF.

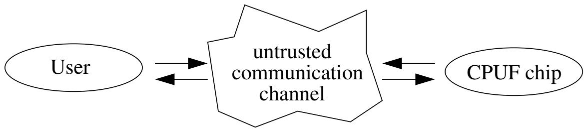

Figure 2: Model for Applications

The responses are only known to the user and the PUF. To establish this property we need a secure way of managing of CRPs as described in section 5.2. CPUFs control the access to CRPs by algorithms which turn out to be the key to secure management. Special attention will be given to protection against man-in-the-middle-attacks while managing CRPs. To prevent man-in-the-middle attacks, we prevent a user from asking for the response to a specific challenge, during the CRP management protocols. This is a concern in the CRP management protocols, as, in these protocols, the chip sends responses to the user. In the application protocols, the responses are used to generate MACs, and are never sent to the user.

5.2 CRP Management Models

In our models for challenge-response pair management, the user does not have CRPs for the CPUF yet, and would like to establish its own private list of CRPs. For challenge-response pair management, we introduce the following 3 new principals:

- *manufacturer*: the manufacturer is the principal that made the chip with the CPUF. When the manufacturer had the chip, and was in physical contact with the chip, it established its own private list of CRPs. We assume that, in the special situation when the manufacturer is in physical contact with the CPUF chip, the communication channel between the manufacturer and the chip is authentic and private. Though the manufacturer was originally in physical contact with the chip, we assume that it does not have the chip now.

- *owner*: the owner is the principal that controls access to the CPUF. The owner has its own private list of CRPs. The owner can be considered to be the principal that bought the CPUF chip from the manufacturer.

- *certifier*: the certifier has its own private list of CRPs for the CPUF, and is trusted by the user. The manufacturer of the CPUF chip can act as a certifier to other users. After the user has established its own private list of CRPs, it may act as a certifier to another user, if the second user trusts the first user. For example, if the user trusts the owner of the chip, the owner of the chip can also act as a certifier.

We have 5 scenarios:

- *bootstrapping*: the manufacturer of the CPUF gets the initial CRP from the CPUF.

- *introduction*: a user, who does not have any CRPs for the CPUF, securely obtains a CRP from a certifier.

- *private renewal*: after obtaining a CRP from a certifier, the user can use this CRP to generate his own private list of CRPs.

- *renewal*: after generating his own private list of CRPs, the user can use one of these to generate more private CRPs.

- *anonymous introduction*: in anonymous introduction, a user, who does not have any CRPs for the CPUF, securely obtains a certified, anonymous, CRP for the CPUF. The user is given a CRP that is certified by the certifier. However, in anonymous introduction, the owner of the CPUF does not want to reveal to the user which CPUF the user is being given a CRP to. Thus, at the end of the protocol, the user knows that he has been given a CRP that is certified by the certifier, and can use this CRP to generate other CRPs with the CPUF and run applications using the CPUF. However, if the user colludes with the certifier, or other users with certified, anonymous CRPs to the CPUF, he will not be able to use the CRPs to determine that he is communicating with the same CPUF as them.

5.2.1 Bootstrapping

Figure 3 illustrates the model for bootstrapping. When a CPUF has just been produced, the manufacturer generates a CRP for it. We assume that, when the manufacturer generates this CRP, it is in physical contact with the chip, and thus, the communication channel is private and authentic. For the other protocols, it is assumed that the manufacturer no longer has the chip.

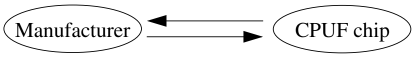

Figure 3: Model for Bootstrapping

5.2.2 Introduction

Figure 4 illustrates the model for CPUF introduction. In introduction, the certifier gives a CRP for the CPUF to the user over a channel that is authentic and private.

As the certifier knows the CRP the user is given, the certifier can read all of the messages the user exchanges with the CPUF using this CRP. The user, thus, needs to use the private renewal protocol to generate his own private list of CRPs.

Furthermore, as, in this scheme, the CPUF honors messages that are MACed with a key generated from the response of the CRP the certifier has given to the user, the user and the certifier can collude to determine that they are communicating with the same CPUF. They, and other users who use the same certifier, may then be able to use this information to track and monitor the CPUF's transactions. The CPUF's owner can introduce the CPUF to the user using the anonymous introduction protocol to deal with this problem.

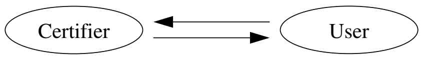

Figure 4: Model for Introduction

5.2.3 Private Renewal

Figure 5 illustrates the model for private renewal. The user is assumed to already have a certified CRP. However, he wants to generate a private list of CRPs. In this model, the communication channel between the user and the CPUF is untrusted.

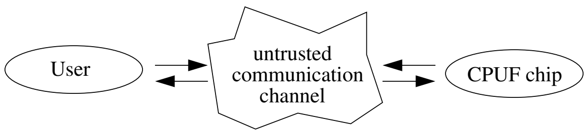

Figure 5: Model for Private Renewal

5.2.4 Renewal

The model for renewal is the same as that for private renewal. The user is assumed to have already generated a private list of CRPs, and would like to generate more private CRPs with the CPUF. He may need more CRPs for his applications, say.

5.2.5 Anonymous Introduction

Figure 6 illustrates the model for anonymous introduction. Again, the user is the principal which does not have CRPs for the CPUF yet, and would like to establish its own private list of CRPs. The communication channels between the certifier, owner and user are secure (private and authentic). The communication channels between each of these principals and the CPUF is untrusted. In our version of the protocol, the certifier and owner communicate with each other, the owner and user communicate with each other, and the owner communicates with the CPUF. The certifier and user can potentially collude to determine if their CRPs are for the same CPUF.

6 Protocols

We will now describe the protocols that are necessary in order to use PUFs. These protocols must be designed to make it impossible to get the response to a chosen challenge. Indeed, if that were possible, then we would be vulnerable to a man-in-the-middle attack that breaks nearly all applications.

6.1 Man-in-the-Middle Attack

Before looking at the protocols, let us have a closer look at man-in-the-middle attack that we must defend against. The ability to prevent this man-in-the-middle attack is *the fundamental difference* between controlled and uncontrolled PUFs.

The scenario is the following. Alice wants to use a challenge-response pair (CRP) that she has to interact with a CPUF in a controlled way (we are assuming that the CRP is the only shared secret between Alice and the CPUF). Oscar, the adversary, has access to the PUF, and has a method that allows him to extract from it the response to a challenge of his choosing. He wants to impersonate the CPUF that Alice wants to interact with.

At some point, in her interaction with the CPUF, Alice will have to give the CPUF the challenge for her CRP so that the CPUF can calculate the response that it is to share with her. Oscar can read this challenge because up to this point in the protocol Alice and the CPUF do not share any secret. Oscar can now get the response to Alice's challenge from the

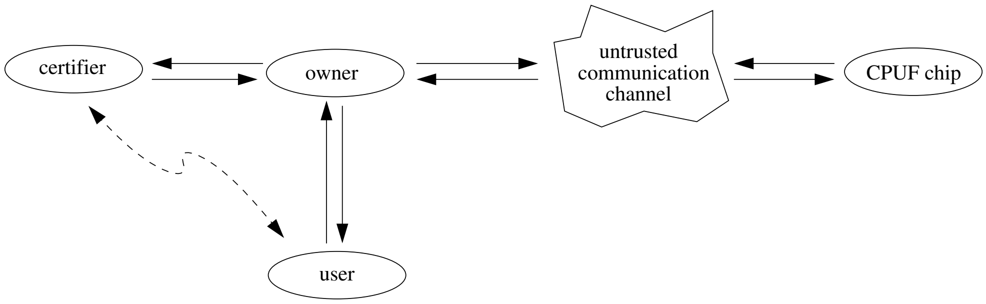

Figure 6: Model for Anonymous Introduction

CPUF, since he has a method of doing so. Once Oscar has the response, he can impersonate the CPUF because he knows everything Alice knows about the PUF. This is not at all what Alice intended.

We should take note that in the above scenario, there *is* one thing that Oscar has proven to Alice. He has proven that he has access to the CPUF. In some applications, such as the key cards from [Rav01], proving that someone has access to the CPUF is probably good enough. However, for more powerful examples such as certified execution that we will cover in section 7.2, where we are trying to protect Alice from the very owner of the CPUF, free access to the PUF is no longer sufficient.

More subtle forms of the man-in-the-middle attack exist. Suppose that Alice wants to use the CPUF to do what we will refer to in section 7.2 as *certified execution*. Essentially, Alice is sending the CPUF a program to execute. This program executes on the CPUF, and uses the shared secret that the CPUF calculates to interact with Alice in a secure way. Here, Oscar can replace Alice's program by a program of his own choosing, and get his program to execute on the CPUF. Oscar's program then uses the shared secret to produce messages that look like the messages that Alice is expecting, but that are in fact forgeries.

6.2 Defeating the Man-in-the-Middle Attack

6.2.1 Basic CPUF Access Primitives

In the rest of this section, we will assume that the CPUF is able to execute some form of program in a private (nobody can see what the program is doing) and authentic (nobody can modify what the program is doing) way. In some CPUF implementations where we do not need the ability to execute arbitrary algorithms, the program's actions might in fact be implemented in hardware or by some other means – the exact implementation details make no difference to the following discussion.

In this paper we will write programs in pseudo-code in which a few basic functions are used:

- `Output(arg1, ...)` is used to send results out of the CPUF. Anything that is sent out of the CPUF is potentially visible to the whole world, except during bootstrapping, where the manufacturer is in physical possession of the CPUF.
- `EncryptAndMAC(message, key)` is used to encrypt and MAC `message` with `key`.
- `PublicEncrypt(message, key)` is used to encrypt `message` with `key`, the public key.
- `MAC(message, key)` MACs `message` with `key`.

The CPUF's control is designed so that the PUF can only be accessed by programs, and only by using two primitive functions: *GetResponse* and *GetSecret*. If f is the PUF, and h is a publicly available collision resistant one-way hash function then the primitives are defined as:

$$\begin{aligned}&\mathit{GetResponse}(\mathit{PreChallenge}) =\\ &\quad f\left(h\left(h\left(\mathit{Program}\right), \mathit{PreChallenge}\right)\right)\\ &\mathit{GetSecret}(\mathit{Challenge}) =\\ &\quad h\left(h\left(\mathit{Program}\right), f\left(\mathit{Challenge}\right)\right)\end{aligned}$$

In these primitives, *Program* is the program that is being run in an authentic way. Just before starting the program, the CPUF calculates $h(\mathit{Program})$, and later uses this value when *GetResponse* and *GetSecret* are invoked. We shall show in the next section that these two primitives are sufficient to implement the CRP management primitives that were detailed in section 5. We shall also see that *GetResponse* is essentially used for CRP generation while *GetSecret* is used by applications that want to produce a shared secret from a CRP.

Figure 7 summarizes the possible ways of going between pre-challenges, challenges, responses and shared secrets. In this diagram moving down is easy.

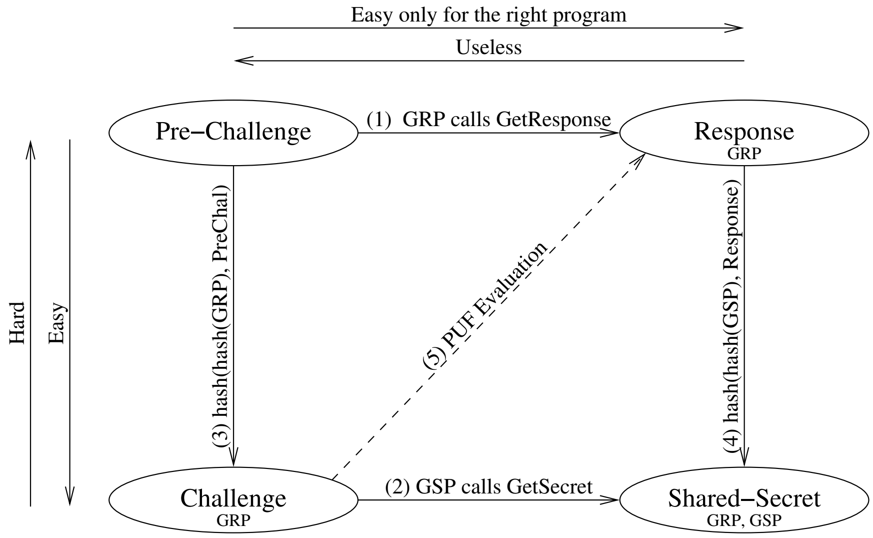

Figure 7: This diagram shows the different ways of moving between Pre-Challenges, Challenges, Responses and Shared-Secrets. The dotted arrow indicates what the PUF does, but since the PUF is controlled, nobody can go along the arrow directly. GRP and GSP are the programs that call *GetResponse* and *GetChallenge* respectively. The challenge and the response depend on the GRP that created them, and the shared secret depends on the GSP.

You just have to calculate a few one-way hashes. Moving up is hard because it would involve reversing those one-way hashes. Going from left to right is easy for the program whose hash is used in the *GetResponse* or *GetSecret* primitives, and hard for all other programs. The difficulty of going from right to left is not important, as the adversary's task wouldn't be easier if it was easy.

6.2.2 Using a CRP to Get a Shared Secret

To show that the man-in-the-middle attack has been defeated, we shall show that a user who has a CRP can use it to establish a shared secret with the PUF (previously, the man-in-the-middle could determine the value of what should have been a shared secret).

The user sends a program like the one below to the CPUF, where `Challenge` is the challenge from the CRP that the user already knows.

```
begin program
  Secret = GetSecret(Challenge);
  /* Program that uses Secret as   *
   * a shared secret with the user */
end program
```

The user can determine `Secret` because he knows the response to `Challenge`, and so he can calculate $h\left(h\left(program\right), response\right)$. Now we must show that a man-in-the-middle cannot determine `Secret`.

By looking at the program that is being sent to the CPUF, the adversary can determine the challenge from the CRP that is being used. This is the only starting point he has to try to find the shared secret. Unfortunately for him, the adversary cannot get anything useful from the challenge. Because the challenge is deduced from the pre-challenge via a one-way function, the adversary cannot get the pre-challenge directly. Getting the Response directly is impossible because the only way to get a response out of the CPUF is starting with a pre-challenge. Therefore, the adversary must get the shared secret directly from the challenge.

However, only a program that hashes to the same value as the user's program can get from the challenge to the secret directly by using *GetSecret* (any other program would get a different secret that can't be used to find out the response or the sought after secret because it is the output of a one-way function). Since the hash function that we are using is collision resistant, the only program that the attacker can use to get the shared secret is the user's program. If the user program is written in such a way that it does not leak the secret to the adversary, then the man-in-the middle attack fails. Of course, it is perfectly possible that the user's program could leak the shared secret if it is badly written. But this is a problem with any secure program, and is not specific to PUFs. Our goal isn't to prevent a program from giving away its secret but to make it possible for a well written program to produce a shared secret.

6.3 Challenge Response Pair Management Protocols

Now we shall see how *GetResponse* and *GetSecret* can be used to implement the key management primitives that were described in section 5.[3] It is worth noting that the CPUF need not preserve any state between program executions.

6.3.1 Bootstrapping

The manufacturer sends the following program to the CPUF, where `PreChallenge` is set to some arbitrary value.

```
begin program
  Response = GetResponse(PreChallenge);
  Output(Response);
end program
```

The user gets the challenge for his newly created CRP by calculating $h(h(program), \texttt{PreChallenge})$, the response is the output of the program.

6.3.2 Renewal

The user sends the following program to the CPUF, where `PreChallenge` is set to some arbitrary value, and `OldChallenge` is the challenge from the CRP that the user already knows.

```
begin program
  NewResponse = GetResponse(PreChallenge);
  Output(EncryptAndMAC(
    NewResponse, GetSecret(OldChallenge)));
end program
```

The user and the CPUF have `GetSecret(OldChallenge)` as a shared secret because knowledge of the initial CRP is needed to produce it. The user can be sure that only he can get `NewResponse`, because it is encrypted with the shared secret. An adversary can change `OldChallenge` to a challenge that he knows the response to, but since `OldChallenge` is part of the program, the newly created CRP would be different from the one that the adversary is trying to hijack (because *GetResponse* combines the pre-challenge with a one-way hash of the program that is being run). The MAC proves that `NewResponse` that the user is getting originated from the CPUF. The user gets the challenge for his newly created CRP by calculating $h(h(program), \texttt{PreChallenge})$.

[3]The implementations that are presented contain the minimum amount to encryption to ensure security. A practical implementation would probably want to include nonces to ensure message freshness, and would encrypt and MAC as much information as possible. In particular, it is not necessary in our model to encrypt the pre-challenges that are used to produce CRPs. Nevertheless hiding the pre-challenge (and therefore the challenge) would make it harder for an adversary to mount an attack in which he manages to forcibly extract the response to a specific challenge from the CPUF.

6.3.3 Introduction

Introduction is particularly easy. The certifier simply sends a CRP to the user over some agreed upon secure channel. In many cases, the certifier will use renewal to generate a new CRP, and then send that to the user. The user will then use private renewal to produce a CRP that the certifier does not know.

6.3.4 Private Renewal

The user sends the following program to the CPUF, where `PreChallenge` is set to some arbitrary value, `OldChallenge` is the challenge from the CRP that the user already knows, and `PubKey` is the user's public key.

```
begin program
  NewResponse = GetResponse(PreChallenge);
  Message =
    PublicEncrypt(NewResponse, PubKey);
  Output(Message,
    MAC(Message, GetSecret(OldChallenge)));
end program
```

The user can be sure that only he can read the `NewResponse`, because it is encrypted with his public key. If the adversary tries to replace `PubKey` by his own public key, he will get the response to a different challenge because `PubKey` is part of the program, and therefore indirectly changes the output of *GetResponse*. The MAC can only be forged by the party that the user is sharing the old CRP with (probably a certifier that the user just performed introduction with). If we assume that that party is not doing an active attack, then we know that the MAC was produced by the CPUF, and therefore, the `NewResponse` is indeed characteristic of the CPUF. The user gets the challenge for his newly created CRP by calculating $h(h(program), \texttt{PreChallenge})$.

6.4 Anonymity Preserving Protocols

In section 4.4 we showed how a CPUF could be made to take on many different personalities in order to preserve the anonymity of its owner. People don't want their CPUF to give away the fact that the same person is gambling on gambling.com and doing anonymous computation for SETI@home. In this section, we shall add a personality selector to the PUF as in figure 1. We shall call the personality selector `PersonalitySel`. The person who is trying to hide his identity will be called the owner of the CPUF, but as we shall see at the end of section 6.4.2 the notion is more general than this. We shall assume that all sources of information concerning the identity of the CPUF's owner have been eliminated by other protocol layers, and shall focus on preventing the CPUF from leaking his identity. We shall also assume that there are enough people using anonymized introduction that traffic analysis (correlating the arrival of a message at a node with the departure of a message a little while later simply from timing considerations) is unusable.

Programs must not be allowed to freely set `PersonalitySel` , or else they could put the CPUF into a known personality and defeat the purpose of having a personality selector. We shall therefore describe how the value of `PersonalitySel` is controlled. First, two new primitive functions are provided by the CPUF:

- `ChangePersonality(Seed)` changes the personality to h(`PersonalitySel`, `Seed`). Where h is a one-way hash function.
- `RunProg(Program)` runs the program that is given as an argument without changing `PersonalitySel`.

Moreover, when a program is loaded into the CPUF from the outside world, and run (as opposed to being run by `RunProg`), `PersonalitySel` is set to zero. We shall call this the default personality.

The pseudo-code uses a few extra primitive functions:

- `Decrypt(mesg, key)` is used to decrypt `mesg` that was encrypted with `key`.
- `HashWithProg(x)` is used to compute $h(h(program),$ `x`). This function reads the area where the CPUF is storing the hash of the program.
- `Hash(...)` is a one-way hash function.
- `Blind(mesg,fact)` is used to apply the blinding factor `fact` to `mesg`. See section 6.4.2 for a brief description of blinding.

6.4.1 Choosing the Current Personality

When the CPUF's owner wants to show a personality other than his CPUF's default personality, he intercepts all programs being sent to the CPUF and encapsulates them in a piece of code of his own:

```
ESeed =
  /* the personality seed  *
   * encrypted with Secret */
EProgram =
  /* the encapsulated  program *
   * encrypted with Secret     */

begin program
  Secret = GetSecret(Challenge);
  Seed = Decrypt(Eseed, Secret);
  Program = Decrypt(EProgram,Secret);

  ChangePersonality(Seed);
  RunProg(Program);
end program
```

In this program, the line that appears before `begin program` is a piece of data that accompanies the program but that does not participate in the hash of the program. If `EProgram` were included in the hash, then we would not be able to encrypt it because the encryption key would depend on the encrypted program. Other values that appear are `Seed`, an arbitrarily selected seed; and `Challenge`, the challenge of one of the owner's CRPs.

By encapsulating the program in this way, the owner is able to change the personality that the CPUF is exhibiting when it runs the user's program. There is no primitive to allow the user's program to see the personality that it is using, and the seed that is used with `ChangePersonality` is encrypted so the user has no way of knowing which personality he is using. The user's program is encrypted, so even by monitoring the owner's communication, the user cannot determine if the program that is being sent to the CPUF is his own program.

6.4.2 Anonymous Introduction

The anonymous introduction protocol is much more complicated than the other protocols we have seen so far. We will only sketch out the details of why it works. This protocol uses blinding, a description of which can be found in [Sch96].

The essential idea of blinding is this: Alice wants Bob to sign a message for her, but she does not want Bob to know what he has signed. To do this Alice hides the message by applying what is called a blinding factor. Bob receives the blinded message, signs it and returns the signed blinded message to Alice. Alice can then remove the blinding factor without damaging Bob's signature. The resulting message is signed by Bob, but if Bob signs many messages, he cannot tell which unblinded message he signed on which occasion.[4]

[4]In this protocol, to avoid over-complication, we have assumed that Alice does not need to know Bob's public key in order to sign a message. For real-world protocols such as the one that David Chaum describes in [Cha85] this is not true. Therefore, an actual implementation of our anonymous introduction protocol might have to include the certifier's public key in the program that is sent to the CPUF. In that case, it should be encrypted to prevent correlation of messages going

Here is the anonymous introduction protocol:

1. The owner collects a challenge from the certifier, and the user's public key. He produces the following program from figure 8 that is sent to the CPUF.

2. The owner decrypts the output from the CPUF, checks the MAC, and passes `Mesg5` on to the certifier, along with a copy of the program (only the part that participates in the MAC) encrypted with the certifier's public key.

3. The certifier decrypts the program, checks that it is the official anonymous introduction program, then hashes it to calculate `CertSecret`. He can then verify that `Mesg4` is authentic with the MAC. He finally signs `Mesg4`, and sends the result to the owner.

4. The owner unblinds the message, and ends up with a signed version of `Mesg3`. He can check the signature, and the MAC in `Mesg3` to make sure that the certifier isn't communicating his identity to the user. He finally sends the unblinded message to the user. This message is in fact a version of `Mesg3` signed by the certifier.

5. The user checks the signature, and decrypts `Mesg2` with his secret key to get a CRP.

Remarks:

- `UserPubKey` and `CertChallenge` must be encrypted, otherwise it is possible to correlate the message that Alice sends to the CPUF with the certifier's challenge or with the user's public key.

- `Seed` must be encrypted to prevent the certifier or the user from knowing how to voluntarily get into the personality that the user is being shown.

- `PreChallengeSeed` must be encrypted to prevent the certifier from finding out the newly created challenge when he inspects the program in step 3.

- The encryption between `Mesg5` and `Mesg6` is needed to prevent correlation of the message from the CPUF to the owner and the message from the owner to the certifier.

Interestingly, we are not limited to one layer of encapsulation. A principal who has gained access to a personality of a CPUF through anonymous introduction can introduce other parties to this PUF. In particular, he can send the signed CRP that he received back to the certifier and get the certifier to act as a certifier for his personality when he anonymously introduces the CPUF to other parties.

to the CPUF with a specific transaction with the certifier.

7 Applications

We believe there are many applications for which CPUFs can be used, and we describe a few here. Other applications can be imagined by studying the literature on secure coprocessors, in particular [Yee94]. We note that the general applications for which this technology can be used include all the applications today in which there is a single symmetric key on the chip.

7.1 Smartcard Authentication

The easiest application to implement is authentication. One widespread application is smartcards. Current smartcards have hidden digital keys that can sometimes be extracted using many different kinds of attacks [And01]. With a unique PUF on the smartcard that can be used to authenticate the chip, a digital key is not required: the smartcard *hardware* is itself the secret key. This key cannot be duplicated, so a person can lose control of it, retrieve it, and continue using it. The smartcard can be turned off if the owner thinks that it is permanently lost by getting the application authority to forget what it knows of the secret signature that is associated with the unique smartcard.

The following basic protocol is an outline of a protocol that a bank could use to authenticate messages from PUF smartcards. This protocol guarantees that the message the bank receives originated from the smartcard. It does not, however authenticate the bearer of the smartcard. Some other means such as a PIN number or biometrics must be used by the smartcard to determine if its bearer is allowed to use it.

1. The bank sends the following program to the smartcard, where `R` is a single use number and `Challenge` is the bank's challenge:

```
begin program
  Secret = GetSecret(Challenge);
  /* The smartcard somehow     *
   * generates Message to send *
   * to the bank               */
  Output(Message, MAC((Message, R), Secret));
end program
```

2. The bank checks the MAC to verify the authenticity and freshness of the message that it gets back from the PUF.

The number `R` is useful in the case where the smartcard has state that is preserved between executions. In that case it is important to ensure the freshness of the message.

```
  /* Various values encrypted with OwnerSecret. */
  ESeed = ...
  EPreChallengeSeed = ...
  EUserPubKey = ...
  ECertChallenge = ...

begin program
  OwnerSecret = GetSecret(OwnerChallenge);
  Seed = Decrypt(ESeed, OwnerSecret);
  PreChallengeSeed = Decrypt(EPreChallengeSeed, OwnerSecret);
  UserPubKey = Decrypt(EUserPubKey, OwnerSecret);
  CertChallenge = Decrypt(ECertChallenge, OwnerSecret);

  CertSecret = GetSecret(CertChallenge);
  PreChallenge = Hash(UserPubKey, PreChallengeSeed);
  NewChallenge = HashWithProg(PreChallenge);
  ChangePersonality(Seed);
  NewResponse = GetResponse(PreChallenge);

  Mesg1 = (NewChallenge, NewResponse);
  Mesg2 = PublicEncrypt(Mesg1, UserPubKey);
  Mesg3 = (Mesg2, MAC(Mesg2, OwnerSecret));
  Mesg4 = Blind(Mesg3, OwnerSecret);
  Mesg5 = (Mesg4, MAC(Mesg4, CertSecret));
  Mesg6 = EncryptAndMAC(Mesg5, OwnerSecret);
  Output(Mesg6);
end program
```

Figure 8: The anonymous introduction program.

If the privacy of the smartcard's message is a requirement, the bank can also encrypt the message with the same key that is used for the MAC.

7.2 Certified execution

At present, computation power is a commodity that undergoes massive waste. Most computer users only use a fraction of their computer's processing power, though they use it in a bursty way, which justifies the constant demand for higher performance. A number of organizations, such as SETI@home and distributed.net, are trying to tap that wasted computing power to carry out large computations in a highly distributed way. This style of computation is unreliable as the person requesting the computation has no way of knowing that it was executed without any tampering.

With chip authentication, it would be possible for a certificate to be produced that proves that a specific computation was carried out on a specific chip. The person requesting the computation can then rely on the trustworthiness of the chip manufacturer who can vouch that he produced the chip, instead of relying on the owner of the chip.

There are two ways in which the system could be used. Either the computation is done directly on the secure chip, either it is done on a faster insecure chip that is being monitored in a highly interactive way by supervisory code on the secure chip.

To illustrate this application, we present a simple example in which the computation is done directly on the chip. A user, Alice, wants to run a computationally expensive program over the weekend on Bob's 128-bit, 300MHz, single-tasking computer. Bob's computer has a single chip, which has a PUF. Alice has already established CRPs with the PUF chip.

1. Alice sends the following program to the CPUF, where `Challenge` is the challenge from her CRP:

   ```
   begin program
     Secret = GetSecret(Challenge);
     /* The certified computation *
      * is performed, the result  *
      * is placed in Result       */
     Output(Result, MAC(Result, Secret));
   end program
   ```

2. The bank checks the MAC to verify the authen-

ticity of the message that it gets back from the PUF.

Unlike the smartcard application, we did not include a single use random number in this protocol. This is because we are assuming that we are doing pure computation that cannot become stale (any day we run the same computation it will give the same result).

In this application, Alice is trusting that the chip in Bob's computer performs the computation correctly. This is easier to ensure if all the resources used to perform the computation (memory, CPU, etc.) are on the PUF chip, and included in the PUF characterization. We are currently researching and designing more sophisticated architectures in which the PUF chip can securely utilize off-chip resources using some ideas from [LTM+00].

There is also the possibility of a PUF chip using the capabilities of other networked PUF chips and devices using certified executions. The PUF would have CRPs for each of the computers it would be using, and perform computations using protocols similar to the one described in this section.

7.3 Software licensing

We are exploring ways in which a piece of code could be made to run only on a chip that has a specific identity defined by a PUF. In this way, pirated code would fail to run. One method that we are considering is to encrypt the code using the PUF's responses on an instruction per instruction basis. The instructions would be decrypted inside of the PUF chip, and could only be decrypted by the intended chip. As the operating system and off-chip storage is untrustworthy, special architectural support will be needed to protect the intellectual property as in [LTM+00].

8 Conclusion

We have described how controlled physical unknown functions (CPUFs) can be applied to two different security problems in this paper.

CPUFs hold promise in creating smartcards with an unprecedented level of security. CPUFs also enable these smartcards or other processors to run user programs in a secure manner, producing a certificate that gives the user confidence in the results generated. While we have not described software licensing and intellectual property protection applications in this paper, the protocols for these applications will have some similarity to those described herein, and are a subject of ongoing work.

References

[And01] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* John Wiley and Sons, 2001.

[Cha85] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28:1030–1040, 1985.

[GCvDD02] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical unknown functions. Technical report, MIT LCS TR-833, May 2002.

[LTM+00] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 169–177, November 2000.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[Rav01] P. S. Ravikanth. *Physical One-Way Functions.* PhD thesis, Massachusetts Institute of Technology, 2001.

[Sch96] Bruce Schneier. *Applied Cryptography.* Wiley, 1996.

[SW99] S. W. Smith and S. H. Weingart. Building a High-Performance, Programmable Secure Coprocessor. In *Computer Networks (Special Issue on Computer Network Security)*, volume 31, pages 831–860, April 1999.

[Yee94] Bennet S. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, 1994.